Rest Services (JAX-RS):

=========================

REST stands for REpresentational State Transfer.

REST is an architectural style not a protocol. It is developed by Roy Thomas Fielding.

JAX-RS: Java API for RESTful Web Services

JAX-RS Implementation:

=======================

1. Jersey

2. RESTEasy

Advantages of RESTful Web Services:

====================================

Fast: RESTful Web Services are fast because there is no strict specification like SOAP. It consumes less bandwidth and resource.

Language and Platform independent: RESTful web services can be written in any programming language and executed in any platform.

Can use SOAP: RESTful web services can use SOAP web services as the implementation.

Permits different data format: RESTful web service permits different data format such as Plain Text, HTML, XML and JSON

HTTP Methods:

================

GET: It reads a resource.

PUT: It updates an existing resource.

POST: It creates a new resource.

DELETE: It deletes the resource.

JAX-RS Annotations:

====================

@GET

Annotate your Get request methods with @GET.

```
@GET
public String getHTML() {
  ...
}
```

@Produces:

@Produces annotation specifies the type of output this method (or web service) will produce.

```
@GET
@Produces("application/xml")
public Contact getXML() {
  ...
}
```

@Path

@Path annotation specify the URL path on which this method will be invoked.

```
@GET
@Produces("application/xml")
@Path("xml/{firstName}")
public Contact getXML() {
  ...
}
```

@PathParam

We can bind REST-style URL parameters to method arguments

```
@GET
@Produces("application/xml")
@Path("xml/{firstName}")
public Contact getXML(@PathParam("firstName") String firstName) {
  Contact contact = contactService.findByFirstName(firstName);
  return contact;
}
```

@QueryParam:

Request parameters in query string can be accessed using @QueryParam annotation

```
@GET
@Produces("application/json")
@Path("json/companyList")
public CompanyList getJSON(@QueryParam("start") int start, @QueryParam("limit") int limit) {
  CompanyList list = new CompanyList(companyService.listCompanies(start, limit));
  return list;
}
```

@POST:

Annotate POST request methods with @POST.

```
@POST

@Consumes("application/json")

@Produces("application/json")

public RestResponse<Contact> create(Contact contact) {

...

}
```

@Consumes:

The @Consumes annotation is used to specify the MIME media types a REST resource can consume.:

```
@PUT

@Consumes("application/json")

@Produces("application/json")

@Path("{contactId}")

public RestResponse<Contact> update(Contact contact) {

...

}
```

@FormParam:

The REST resources will usually consume XML/JSON for the complete Entity Bean. Sometimes, you may want to read parameters sent in POST requests directly and you can do that using @FormParam annotation. GET Request query parameters can be accessed using @QueryParam annotation.

```
@POST

public String save(@FormParam("firstName") String firstName,
    @FormParam("lastName") String lastName) {
      ...
  }
```

@PUT:

Annotate PUT request methods with @PUT. This is used to update the existing resource.

```
@PUT
@Consumes("application/json")
@Produces("application/json")
@Path("{contactId}")
public RestResponse<Contact> update(Contact contact) {
...
}
```

@OPTIONS:

In REST OPTIONS is a method level annotation, this annotation indicates that the following method will respond to the HTTP OPTIONS request only. It is used to request, for information about the communication option available for a resource.

```
@OPTIONS
  @Produces(MediaType.APPLICATION_JSON)
  @Path("/")
  public Response optionsForBookResource() {
      return Response.status(200)
        .header("Allow","POST, PUT, GET")
        .header("Content-Type", MediaType.APPLICATION_JSON)
        .header("Content-Length", "0")
        .build();
  }
```

This method allows the client of the REST API to determine, which HTTP method (GET, HEAD, POST, PUT, DELETE) can be used for a resource identified by requested URI, without initiating a resource request by using any particular HTTP method. Response to this method are not cacheable.

HATEOAS:

===========

HATEOAS acronyms for Hypermedia as the Engine of Application State.The term hypermedia refers to content that contains a link to other forms of media like images, movies, and text.sing HATEOAS, a client interacts with a network application, whose application server provides information dynamically through Hypermedia.

In the Spring HATEOAS project, we do not require Servlet Context and concatenate the path variable to the base URI. Instead of this, Spring HATEOAS offers three abstractions for creating the URI: ContrrollerLinkBuilder, Link, and Resource Support. We can use these abstractions to create metadata, which associates with the resource representation.

Suppose, we have requested a GET request for localhost:8080/users/1, it returns the details of user id 1. Along with this, it also returns a field called link that contains a link (localhost:8080/users) of all users so that consumers can retrieve all the users. This concept is called HATEOAS.

Example:

============

```
Resource<User> resource=new Resource<User>(user);   //constructor of Resource class

//add link to retrieve all the users

ControllerLinkBuilder linkTo=linkTo(methodOn(this.getClass()).retriveAllUsers());

resource.add(linkTo.withRel("all-users"));
```

Internationalization of RESTful Services:

==

Internationalization is the process of designing web applications or services in such a way that it can provide support for various countries, various languages automatically without making the changes in the application. It is also known as I18N because the word internationalization has total 18 characters starting from I to N.

```
@Bean

public  LocaleResolver localeResolver()

{

SessionLocaleResolver localeResolver = new SessionLocaleResolver();

localeResolver.setDefaultLocale(Locale.US);

return localeResolver;

}
```

Content Negotiation:

===========================

Content negotiation is the process of selecting the best representation for a given response when there are multiple representations available. It is a part of HTTP that makes it possible to serve different versions of a document at the same URI.

Content Negotiation Implementing Support for XML

In this section, we will discuss another concept of RESTful Web Services that is content negotiation.

Content Negotiation

A resource can have many representations, mostly because there may be multiple clients expecting different representations.

Content negotiation is the process of selecting the best representation for a given response when there are multiple representations available. It is a part of HTTP that makes it possible to serve different versions of a document at the same URI.

In web API, content negotiation is performed at the server side to determine the media type formatted to be used based on return the response for an incoming request from the client-side. Content negotiation is centered on the media type and media type formatter.

Content negotiation using HTTP headers:

An incoming request may have an entity attached to it. To determine the type of entity server uses the HTTP request header Content-Type. There are some common content types are: application/json, application/xml, text/html, images/jpg, etc.

When a consumer sends a request, it can specify two HTTP Headers related to Content Negotiation

Accept and

Content-Type

Content-Type indicates the content type of the body of the request.

Accept indicates the expected content type of the response.

For example, if a consumer sends a request to http://localhost:8080/students/10001 with Accept header as 'application/xml', we need to provide the xml representation of the resource.

```
<Student>

   <id>10001</id>

   <name>Ranga</name>

   <passportNumber>E1234567</passportNumber>

</Student>
```

If a consumer sends a request with Accept header as 'application/json', we need to provide the JSON representation of the resource.

```
{

  "id": 10001,

  "name": "Ranga",

  "passportNumber": "E1234567"

}
```

Similar concept applies to the Response Body Content based on the Content-Type.

A consumer can send a POST request to http://localhost:8080/students with Content-Type header as ‘application/xml’, and provide the XML representation of the resource to be created.

```
<Student>
   <name>Ranga</name>
   <passportNumber>E1234567</passportNumber>
</Student>
```

A consumer can also send a POST request to http://localhost:8080/students with Content-Type header as ‘application/json’, and provide the JSON representation of the resource to be created.

```
{
  "name": "Ranga",
  "passportNumber": "E1234567"
}
```

Content negotiation using URL patterns

--

There is another way to pass content type information to the server. The client can use a specific extension in resource URIs. For example, a client can request for the following:

http://demo.com/product/mobile/samsung/galaxy-s8/functions.xml

http://demo.com/product/mobile/samsung/galaxy-s8/functions.json

Richardson Maturity Model

==========================

The Richardson maturity model is a way to grade your API according to the constraints of REST. It breaks down the principal element of the REST approach into four levels (0 to 3).

Level 0: The Swamp of POX (Plain Old XML):

To get and post the data, we send a request to the same URI, and only the POST method may be used. These APIs use only one URI and one HTTP method called POST. In short, it exposes SOAP web services in the REST style.

For example, there can be many customers for a particular company. For all the different customers, we have only one endpoint. To do any of the operations like get, delete, update, we use the same POST method.

To get the data: http://localhost:8080/customer

To post the data: http://localhost:8080/customer

Single URI + Single HTTP methods

Level 1: Resources:

It uses multiple URIs. Where every URI is the entry point to a specific resource. It exposes resources with proper URI. Level 1 tackles complexity by breaking down huge service endpoints into multiple different endpoints. It also uses only one HTTP method POST for retrieving and creating data.

For example, if we want a list of specific products, we go through the URI http://localhost:8080/products . If we want a specific product, we go through the URI http://localhost:8080/products/mobile

Many URI + Single HTTP methods

Level 2: HTTP Verbs

At level 2, correct HTTP verbs are used with each request. It suggests that in order to be truly RESTful, HTTP verbs must be used in API. For each of those requests, the correct HTTP response code is provided.

We don't use a single POST method for all requests. We use the GET method when we request a resource, and use the DELETE method when we want to delete a resource. Also, use the response codes of the application protocol.

Many URI + Many HTTP methods

Level 3: Hypermedia Controls

--

Level 3 is the highest level. It is the combination of level 2 and HATEOAS. It also provides support for HATEOAS. It is helpful in self-documentation.

Many URI + Many HTTP methods + HATEOAS.

Rest services HTTP Status codes:

==================================

HTTP defines these standard status codes that can be used to convey the results of a client's request.

1xx: Informational – Communicates transfer protocol-level information.

2xx: Success – Indicates that the client's request was accepted successfully.

3xx: Redirection – Indicates that the client must take some additional action in order to complete their request.

4xx: Client Error – This category of error status codes points the finger at clients.

5xx: Server Error – The server takes responsibility for these error status codes.

Rest Template:

======================

Rest Template used consume RESTful Web Services.

Get Methods:

----

getForObject(url, classType) – retrieve a representation by doing a GET on the URL. The response (if any) is unmarshalled to given class type and returned.

getForEntity(url, responseType) – retrieve a representation as ResponseEntity by doing a GET on the URL.

exchange(requestEntity, responseType) – execute the specified RequestEntity and return the response as ResponseEntity.

execute(url, httpMethod, requestCallback, responseExtractor) – execute the httpMethod to the given URI template, preparing the request with the RequestCallback, and reading the response with a ResponseExtractor.

Post Methods:

----

postForObject(url, request, classType) – POSTs the given object to the URL, and returns the representation found in the response as given class type.

postForEntity(url, request, responseType) – POSTs the given object to the URL, and returns the response as ResponseEntity.

postForLocation(url, request, responseType) – POSTs the given object to the URL, and returns returns the value of the Location header.

exchange(url, requestEntity, responseType)

execute(url, httpMethod, requestCallback, responseExtractor)

Delete methods:

----

delete(url) – deletes the resource at the specified URL.